

UNIVERSITY OF ILLINOIS-URBANA

510.84 IL6R no. C002 no. 916-920(1978)

Continued-product algorithms for division



3 0112 088403727

A PATH PASCAL LANGUAGE

by

Roy H. Campbell

Thomas J. Miller

April 20, 1978



DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

The Library of the

MAY 4 1978

University of Illinois
at Champaign



Digitized by the Internet Archive
in 2013

<http://archive.org/details/pathpascallangua919camp>

A PATH PASCAL LANGUAGE

Roy Harold Campbell
Thomas John Miller

Department of Computer Science

University of Illinois at Champaign-Urbana

Urbana, Illinois 61801

217-333-0215

April 14, 1978

ABSTRACT

This paper describes an implementation of Open Path expressions in the programming language Pascal. The extended language is being used to gain experience with path expressions in the design and construction of practical real time systems and operating systems. The extended language includes an encapsulation mechanism and a novel technique which associates path expressions with access right exportation.

1 Introduction.

Aerospace computer systems are becoming more complex and more dependent on the design and construction of system software that provides asynchronous processes, synchronization, coordination and communication between processes, and guaranteed performance within a set of real time constraints. Such design is expensive and error prone. Our research group is investigating and developing various mechanisms to aid the design and construction of such software for use in the flight research programs undertaken by NASA.

We are investigating whether the structuring facility provided by path expressions will be of practical use in programming real time systems and operating systems. Path expressions were introduced as a technique for specifying process synchronization by [Campbell and Habermann,74], and further discussed by [Habermann,75], [Lauer and Campbell,75], [Flon and Habermann,76], and [Campbell,77]. Variations of the path expression idea have been proposed by [ONERA CERT,77] and notations that are similar to paths that model system behavior have been developed independently by [Shaw,77] and [Riddle,76]. There is much current interest in the design of high level system programming languages that include synchronization mechanisms [Fisher,78].

We designed Path Pascal to help us understand the problems that arise when path expressions are combined with a language to provide a system programming tool. We modified an implemented language rather

than design our own because our main research interests and efforts were in the specific issues of synchronization and real time programming, not language design. Instead of modifying the base language extensively, we elected to add a minimal number of features such that programs in the base language could still be compiled and would still execute in their original manner.

The modifications to the base language involved adding processes, Open Path expressions [Campbell,77], and an encapsulation mechanism. The encapsulation mechanism integrated open paths with a method to export synchronized access to shared, protected data. This is the first practical implementation of Open Paths, and the method associating path expressions with access right exportation is an interesting new technique. The following sections explain the reasons for choosing Pascal [Jensen and Wirth,75] as our base language, Open Path notation, our encapsulation mechanism, and our implementation of processes.

2 The Language

2.1 Base Language

Pascal was adopted as the base language for the implementation of Path Expressions for several important reasons. Pascal has been successfully used as a base for two existing system languages

(Concurrent Pascal [Brinch Hansen,77] and Modula [Wirth,77]) and has been proposed as the foundation for the final Department of Defense language [Fisher,78]. Pascal has been adopted by NASA for use as a software development tool. Finally, we chose to work with the Pascal P compiler [Ammann, Nori, and Jacobi,76] because it is portable and provides a small and easily modified compiler and run time implementation. An informal introduction to the notation may be obtained by examining the examples in Section 2.5.

2.2 Open Path Notation

Campbell examined several path notations in his Ph.D. thesis [Campbell,77], some of which are Elementary Paths, Regular Paths, and General Paths. We chose to implement Open Paths in Path Pascal because they are more efficient and flexible than the notation presented in [Campbell and Habermann,74].

An open path expression specifies the synchronization constraints for a set of procedure executions, some of which may be concurrent. The programming of synchronization with this mechanism is different from the approach used for path notations based upon regular expressions [Habermann,75], [Campbell and Lauer,75], [Campbell,77]. The occurrence of a procedure name in an open path does not necessarily impose any synchronization upon executions of that procedure by processes. As we shall show, Open Paths may also be integrated with encapsulation mechanisms in a novel and practical way.

2.3 Syntax of Open Paths

The syntax of Open Paths is expressed using the BNF formalism presented below.

```

<open path> ::= PATH <list> END;
<list> ::= <sequence> | <sequence> , <list>
<sequence> ::= <item> | <item> ; <sequence>
<item> ::= <unsigned integer> : (<list>)
          | [<list>]
          | <procedure id>

```

2.4 Semantics and Reduction Algorithm

An open path can be implemented using P and V operations on counting semaphores in the prologues and epilogues of the procedures, functions, and processes which are named in that path expression. The following recursive algorithm [Campbell,77] will translate open paths into this implementation. In general, the path expression to be translated will be surrounded by two generated synchronization operations L and R which are on its left and right respectively. Apply the following set of transformation rules in the order given by a parse of the path expression using the production rules above.

A. Replace:- L sequence, list R by:-

```

          L sequence R
and
          L list R

```

The comma is used as a distributive mechanism for the synchronization in which it is embedded.

B. Replace:- L item;sequence R by:-

L item V(s1)
and
P(s1) sequence R

The semicolon is used as a sequencing mechanism. Initialize semaphore s1 to 0.

C. Replace:- L n:(list) R by:-

P(s2) L list R V(s2)

This construction permits up to n simultaneous executions by processes of the procedures, functions, and process instantiations in the list. s2 is initialized to n.

D. Replace:- L [list] R by:-

PP(c,s,L) list VV(c,s,R)

where PP and VV are defined as below:

```
PROCEDURE PP (COUNTER c; SEMAPHORE s; PROCEDURE synch);
BEGIN
  P(s);
  c := c + 1;
  IF c=1 THEN synch;
  V(s)
END;

PROCEDURE VV (COUNTER c; SEMAPHORE s; PROCEDURE synch);
BEGIN
```

```

P(s);
c := c - 1;
IF c=0 THEN synch;
V(s)
END;

```

The PP and VV operations implement the simultaneous execution synchronization. Initialize semaphore s to 1 and counter c to 0.

E. Replace:- L procedurename R by:-

```

      L
body of procedure
      R

```

This algorithm restricts a procedure, function, or process name from appearing more than once in a given Open Path expression. This restriction is easy to remove if repeated names are treated as further nesting of synchronization constraints. We await practical results before determining whether this restriction should be removed.

2.5 Examples of Simple Open Paths

It is beneficial if we give simple examples of how process synchronization may be specified using Open Paths.

1. PATH write END;

```
PROCEDURE write;
  BEGIN
    body
  END;
```

This path provides no synchronization at all and any number of processes may access the procedure write at any time.

2. PATH 2:(write) END;

```
PROCEDURE write;
  BEGIN
    P(s0);
    body
    V(s0)
  END;
```

At most two processes at a time may be executing the procedure write. The semaphore is initialized to 2.

3. PATH write;read END;

PROCEDURE write;	PROCEDURE read;
BEGIN	BEGIN
body	P(s0);
V(s0)	body
END;	END;

This path expressions states that we may have any number of processes executing the write procedure but only as many processes executing the read procedure as have finished the write procedure. The semaphore s0 is initialized to 0.

4. PATH 1:(write,read) END;

PROCEDURE write;	PROCEDURE read;
BEGIN	BEGIN
P(s0);	P(s0);
body	body
V(s0)	V(s0)
END;	END;

This path expression states that either a process may execute the write procedure or a process may execute the read procedure. The semaphore s0 is initialized to 1.

5. Path 5:(write;read) END;

PROCEDURE write;	PROCEDURE read;
BEGIN	BEGIN
P(s0);	P(s1);
body	body
V(s1)	V(s0)
END;	END;

Up to 5 writes may be attempted before a read may be attempted. A read may only start when at least 1 write has completed. Semaphore s0 is initialized to 5 and semaphore s1 is initialized to 0.

6. Path 2:(1:(write);1:(read)) end;

PROCEDURE write;	PROCEDURE read;
BEGIN	BEGIN
P(s0);	P(s1);
P(s2);	P(s3);
body	body
V(s3);	V(s2);
V(s0)	V(s1)
END;	END;

When nested synchronization constructions appear in the path expression, the synchronization that is expressed may be determined by evaluating the inner most synchronzation first and then working outwards. In this path expression write operations execute mutually exclusively as do reads. There can be at most two processes executing reads and writes. A read may execute concurrently with a write provided all other synchronization considerations have been met. Semaphores s0 and s1 are initialized to 1, s2 is initialized to 2 and s3 is initialized to 0.

7. PATH 1:(write;[read]) END;

PROCEDURE write;	PROCEDURE read;
BEGIN	BEGIN
P(s0);	PP(c,s,P(s1));
body	body
V(s1)	VV(c,s,V(s0))
END;	END;

The simultaneous execution construction allows concurrent execution of an operation to be synchronized with respect to other operations. In the above case, read executions may overlap other read executions, but write executions may not overlap other write or read executions. Reading, once started, will continue as long as there are processes invoking read and at least one process is executing read. S and s0 are initialized to 1 and s1 is initialized to 0.

2.6 Encapsulation

The choice of encapsulation mechanisms was influenced by the Pascal Algol-like scope rules. Encapsulation has been included in several Pascal based languages, three of which are Concurrent Pascal, Modula, and Euclid [Lampson et al, 77]. Our implementation of Path Pascal allows correct compilation of standard Pascal programs and provides the necessary protection for the synchronization techniques. Our mechanism may lack the generality of other mechanisms, but it is similar to the facilities provided by Concurrent Pascal. We decided that the use of modules as in Modula or other similar proposed encapsulation techniques would not provide a sufficiently significant advantage in programming compared to the modifications that would be required to the Pascal compiler and language and our approach of combining path expressions with encapsulation. Similarly, we decided that it was inappropriate at this time to devise new encapsulation mechanisms without first gaining practical experience with the problems and advantages of programming using this simple scheme.

```
<object type> ::= OBJECT
    PATH <list> END;
    <type declaration part>
    <variable declaration part>
    <local and entry procedure,
        function and process part>
    <initialization part>
END
```

The encapsulation construction in Path Pascal is called an object and is implemented as an extension of the Pascal structured type facility. Objects satisfy the requirement of a data structuring mechanism described in [Campbell and Habermann,74]. Objects may be declared with an explicit name in a type declaration and variables may be declared of that type. Alternatively, variables may be declared to be an object with an implicit name. An object is used to define a data abstraction and the operations which may be performed on that abstraction. A variable declaration of the object introduces an instance upon which the set of operations may be performed.

```
<entry procedure declaration part> ::= ENTRY PROCEDURE <identifier>
                                   <formal parameter part> <block>
```

The operations which may be performed on an object are entry procedure and entry function invocations and entry process initializations. All the operations on the object must be synchronized explicitly by the path expression for that object. Path Pascal checks at compile time that every operation specified within the path expression has a definition within the object body.

The object body also includes type and variable declarations which constitute the implementation of that object, together with additional local procedures, functions and processes. These, however, are inaccessible outside of the object body. Within the body of the object, the standard Pascal scope rules apply. Procedures and functions within

the body of an object may be recursive. Recursive entry procedures or entry functions may result in deadlock which may be not be immediately detected by the present run time implementation.

<initialization part> ::= INIT; <block>

An object may include an optional initialization procedure which is invoked to initialize the variables that are local to an instance of that object. The initialization procedure is executed on entry to the block which declares that instance. Structured types containing objects may be declared, however assignment between variables of such types is not permitted. Similarly, the Pascal NEW function is not defined for records containing objects. Variables that are objects or contain objects are passed as reference parameters to procedures, functions and processes. Recursive object definitions cause a compile time error.

The simple circular buffer example below illustrates some of the features of Path Pascal. The example is taken from an earlier paper on path expressions [Campbell and Habermann,74] for purposes of comparison.

```

1 TYPE
2   buffer = OBJECT
3     PATH 5 : (:(fill);:(empty) ) END;
4     TYPE
5       buffsize = 0..4;
6       inbuff = ARRAY [buffsize] of CHAR;
7       nextspot = OBJECT
8         PATH 1 : ( inpointer ) , 1 : ( outpointer ) END;
9       VAR
10        in,out : buffsize;
11        ENTRY PROCEDURE inpointer( VAR x : buffsize );

```

```

12         BEGIN
13             x := ( in + 1 ) MOD 5;
14             in := x
15         END;
16     ENTRY FUNCTION outpointer : buffsize;
17     BEGIN
18         out := ( out + 1 ) MOD 5;
19         outpointer := out
20     END;
21     INIT;
22     BEGIN
23         in := 0; out := 0
24     END;
25     END; (*nextspot*)
26     VAR
27         buf : inbuf;
28         pointers : nextspot;
29     ENTRY PROCEDURE fill ( inchar : CHAR );
30     VAR
31         x : buffsize;
32     BEGIN
33         pointers.inpointer(x);
34         buf[x] := inchar
35     END; (*fill*)
36     ENTRY FUNCTION empty : CHAR;
37     VAR
38         x : buffsize;
39     BEGIN
40         x := pointers.outpointer;
41         empty := buf[x]
42     END;
43     END; (*buffer*)

```

This example illustrates circular buffering using five buffers. The path in line 3 states that 5 'fill' operations might be attempted before one 'empty' operation, but no 'empty' operation may begin until at least one 'fill' has been completed. Processes invoking the 'fill' and 'empty' routines must know which buffer they should access. This information is provided by the nested object, 'nextspot'. Only one process at a time should be able to access either of the buffer pointers

('in' and 'out'), but one process may access the 'in' pointer while another process is accessing the 'out' pointer. These synchronization constraints are imposed by the path expression in line 8. Note, an alternative scheme might have been to declare two separate variables to represent the buffer pointers. Lines 21-24 show how it is possible to initialize object variables at block entry time. Lines 33 and 40 show the mechanism required to access object variables. This notation is inspired by the Simula 67 dot notation.

2.7 Processes

Process handling is an interesting aspect of the Path Pascal language. Each process has a run time heap and stack. Storage for an instantiation of a process is obtained from the heap of the process which is performing the instantiation. Processes may be created dynamically and, unlike Modula or Concurrent Pascal, their creation is not restricted to the main program block. Allocated storage for processes may be released and reused after process completion, the storage behaving like dynamic variable storage. Releasing process storage before process completion is an error and is equivalent to releasing dynamic variable storage prior to reaccessing that variable. Processes may have their stack storage requirements estimated using the size attribute, which if not specified, assumes a default value.

Errata

Page 14, line 2 should read

These synchronization constraints are imposed by the path expression in line 3 and redundantly in the path expression in line 8. For efficiency, line 8 may be rewritten as:-

```
PATH inpointer , outpointer END;
```



```

<process declaration> ::= PROCESS <identifier> <size part>
                           <formal parameter part> <block>

```

Instances of a process are created by using the process name as in procedure invocations. The lifetime of any block containing process declarations is greater than or equal to the lifetime of an instantiation of those processes. We have not provided any mechanism to terminate processes abnormally. Processes terminate only when they reach the end of their code body. Any block containing an instantiation of a process declared within a surrounding scope may complete before the process completes.

Processes are declared in the same fashion as procedures are declared in standard Pascal. Processes may take parameters. Actual parameters passed to a process may be either values or, if passed by reference, must have a static scope greater than or equal to that of the process. This restriction is enforced by the compiler to prevent references to non-existent variables which might arise when a procedure, function, or process calls a process and completes before the called process completes.

EXAMPLE

```

1 PROCESS printer ( VAR inbuff : buffer);
2   VAR outbuff : buffer; ch : CHAR;
3
4   PROCEDURE asciitoebcdic ( ch : CHAR);
5     BEGIN
6       .... outbuff.fill(ch)

```

```

7      END;
8
9      PROCESS reader;
10     VAR ch : CHAR;
11     BEGIN
12         REPEAT
13             ch := inbuff.empty;
14             asciitoebcdic(ch)
15         UNTIL false
16     END;
17
18 BEGIN
19     reader;
20     REPEAT    (* simulate a writer *)
21         ch := outbuff.empty;
22         write(ch)
23     UNTIL false
24 END;    (* printer *)

```

This example illustrates processes in Path Pascal. A printer with one parameter, a variable of type buffer declared earlier, manipulates and then prints sequences of characters. Printer declares an additional buffer (line 2), procedure `asciitoebcdic` (line 4) and process reader (line 9). The reader transforms ascii characters in `inbuff` to ebcidic characters using `asciitoebcdic`. The ebcidic characters are placed in `outbuff` and then printed using the write infinite loop.

To experiment with real time programming, we have allowed processes to be associated with a static priority. We have implemented two priority schemes: one in which the priority of all processes is the same, and one in which priorities are determined by the static nesting level of the process declaration. In this latter scheme, processes declared at the outer most levels in a program have highest priority.

The assumption underlying this priority scheme is that processes declared in outer blocks will perform hardware related operations and should be executed with a higher priority. Within a given priority level, a first-in first-out scheduler selects a process for execution. In initial experiments, we found that while this priority scheme could be used to program possible real time systems, it required an unnatural nesting order of processes. The example above uses this priority scheme to ensure that the device driven by routine write (line 22) is kept as busy as possible. This aspect of our project is still under revision and will be modified as we gain experience in working with Path Pascal. We also have the ability to delay processes, again for real time experimentation.

2.8 Further Research and Experiments.

Following the completion of our first experimental Path Pascal compiler, we are now using the language to study real time programming and the problems of systems programming. An assembler is being written to allow us to run Path Pascal programs on a PDP11/20, which will be used in our research. Immediately, we are investigating the possibility of programming interrupt handlers using a mechanism which is similar to the device processes and DOIIO statement of Modula. Eventually, we hope to provide Path expression descriptions of the hardware which will be compiled directly to an interface between software and hardware. In our real time research, we are currently examining the possibility of

specifying deadlines and timing assertions within a program and providing compile time mechanisms to test the feasibility of those deadlines. However, this research at present involves modification of several of the Pascal control statements and the procedure invocation mechanism. Our restriction prohibiting assignment between variables of the same object type makes copying of local object variables cumbersome. However, encapsulation techniques based on modules and monitors do not allow instantiation of data structures with their own local synchronization. Further development of encapsulation techniques is required. Finally, research continues into alternative path notations!

3 Conclusion

An important issue in the design of appropriate language mechanisms for synchronization and coordination of concurrent processes is the practical implications of such language features and whether they provide appropriate concepts for actual system work. We are assessing the practical value of path expressions to programmers. We have extended Pascal with processes, objects, and Open Path expressions. These constructions are simple and permit standard Pascal to be compiled. The extended Pascal includes an encapsulation mechanism which resembles a Simula class. We claim that our object construction permits synchronization using path expressions and data abstractions to be cleanly integrated with Pascal and allows a simple implementation of these constructs in the Pascal P compiler. Further, it ensures that all

access to encapsulated data has an explicit synchronization specification, a feature which adds to the comprehensibility of programs. We intend to use this language in experiments in which we shall program example real time systems and operating systems. In this way we believe we can contribute further to the development of appropriate language mechanisms for system programming.

Acknowledgements

We wish to thank T. Schaefges for his contributions to the experimental Path Pascal implementation. This project is funded in part by project MUST in NASA [NSG 1471]. The experiment also provides a base for an NSF funded project [MCS 77-09128] which is investigating the practical applications of path expressions in programming languages.

REFERENCES

- (Ammann, Nori, and Jacobi,76) U. Ammann, K. Nori, and C. Jacobi, The Portable Pascal Compiler. Institut fuer Informatik, EIDG. Technische Hochschule CH-8096 Zuerich.
- (Brinch Hansen,77) P. Brinch Hansen, The Architecture of Concurrent Programs, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1977.
- (Campbell and Habermann,74) R.H. Campbell and A.N. Habermann, The Specification of Process Synchronization by Path Expressions. Lecture Notes in Computer Science (Editor G. Goos and J. Hartmanis), pp. 89-102, V16, Springer Verlag, 1974.
- (Campbell,77) R.H. Campbell, Path Expressions : A technique for specifying process synchronization. Ph.D. Thesis, The University of Newcastle Upon Tyne, August, 1976.
- (Fisher,78) D. Fisher, DoD's Common Programming Language Effort. Computer, pp. 25-33, March 1978.
- (Flon and Habermann,76) L. Flon and A.N. Habermann, Towards the Construction of Verifiable Software Systems, SIGPLAN Notices 8.2, March 1976.

- (Habermann,75) A.N. Habermann, Path Expressions. Carnegie-Mellon Technical Report, 1975.
- (Jensen and Wirth,75) K. Jensen and N. Wirth, Pascal User Manual and Report. Springer-Verlag, New York, 1975.
- (Lampson et al.,77] B.W. Lampson, J.J. Horning, R.L. London, J.G. Mitchell, and G.J. Popek, Report on the Programming Language Euclid. Sigplan Notices, 12, 2, pp. 1-79, 1977.
- (Lauer and Campbell,75) P.E. Lauer and R.H. Campbell, Formal Semantics of a Class of High Level Primitives for Co-ordinating Concurrent Processes. Acta Informatica, 5, pp. 297-332, 1975.
- (ONERA CERT,78) Parallelism, Control and Synchronization Expression in a Single Assignment Language. Sigplan Notices 13,1, January 1978.
- (Riddle,76) W.E. Riddle, Software System Modelling and Analysis. RSSM/25, Tech. Report, Department of Computer and Communication Sciences, University of Michigan, July, 1976.
- (Shaw,77) A.C. Shaw, Software Descriptions with Flow Expressions. Department of Computer Science, University of Washington, October, 1977.
- (Wirth,77) N. Wirth, Modula: a Language for Modular Multiprogramming. Software-Practice and Experience, 7, pp. 3-84, 1977.

BIBLIOGRAPHIC DATA SHEET	1. Report No. UIUCDCS-R-78-919	2.	3. Recipient's Accession No.
Title and Subtitle A PATH PASCAL LANGUAGE			5. Report Date April 20, 1978
			6.
Author(s) Roy H. Campbell and Thomas J. Miller			8. Performing Organization Rept. No.
Performing Organization Name and Address Department of Computer Science University of Illinois at Urbana-Champaign Urbana, Illinois 61801			10. Project/Task/Work Unit No.
			11. Contract/Grant No. NASA NSG 1471
Sponsoring Organization Name and Address National Aeronautics and Space Administration Langley Research Center Hampton, Virginia 23665			13. Type of Report & Period Covered
			14.

Supplementary Notes

Abstracts

This paper describes an implementation of Open Path expressions in the programming language Pascal. The extended language is being used to gain experience with path expressions in the design and construction of practical real time systems and operating systems. The extended language includes an encapsulation mechanism and a novel technique which associates path expressions with access right exportation.

Key Words and Document Analysis. 17a. Descriptors

encapsulation, object, path expressions, concurrent processes
P and V operations, real time programming, types

Identifiers/Open-Ended Terms

COSATI Field/Group

Availability Statement

19. Security Class (This Report)
UNCLASSIFIED

21. No. of Pages

20. Security Class (This Page)
UNCLASSIFIED

22. Price



